Compilers

Dr. Sherin ElGokhy

Lecture#8

Bottom-Up Parsing

Outline

- Bottom-Up Parsing
- Shift-Reduce Parsing
- Handles
- Recognizing Handles

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method, that is used in most of the parser generated tools.

An Introductory Example

- Bottom-up parsers don't need left-factored grammars
- Revert to the "natural" grammar for our example:

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

- Consider the string: int * int + int

The Idea

Bottom-up parsing **reduces** a string to the start symbol by **inverting productions:**

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

Observation

- Read the productions in reverse (from bottom to top)
- Trace a rightmost derivation!

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

By using reductions instead of productions

A Bottom-up Parse

int * int + int

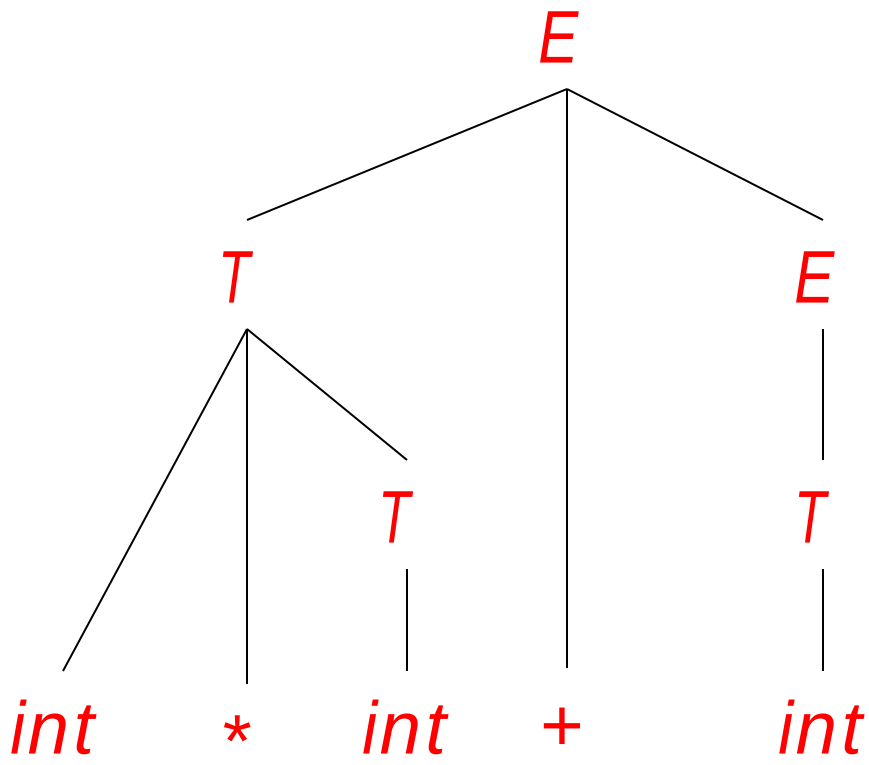
int * T + int

T + int

T + T

T + E

E



Parse tree constructed from the given sequence of reductions

A Bottom-up Parse in Detail (1)

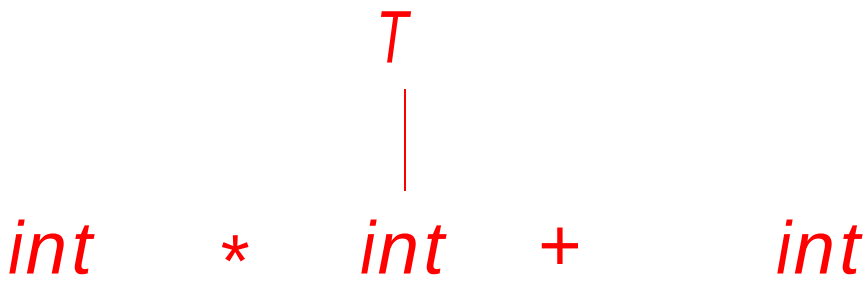
int * int + int

int * *int* + *int*

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

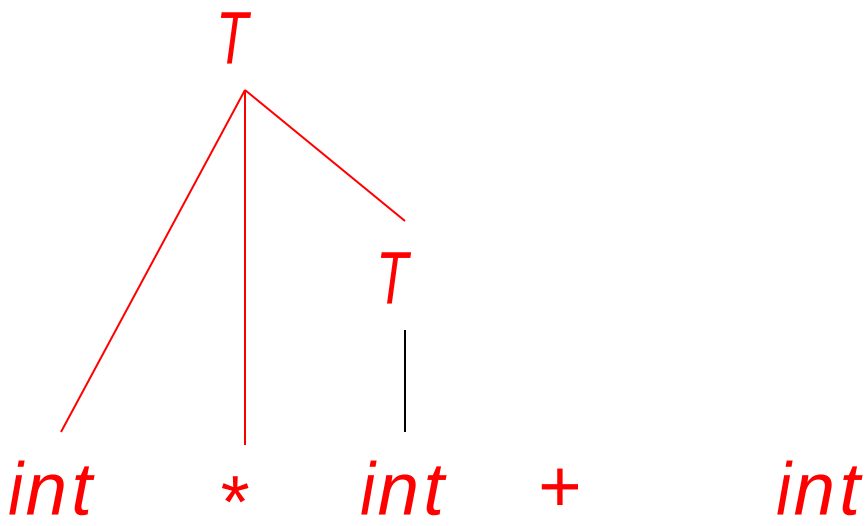


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



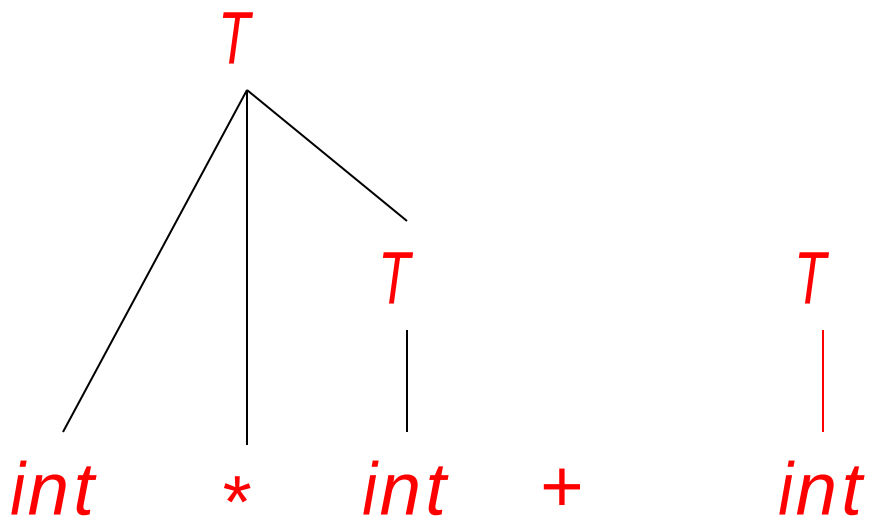
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

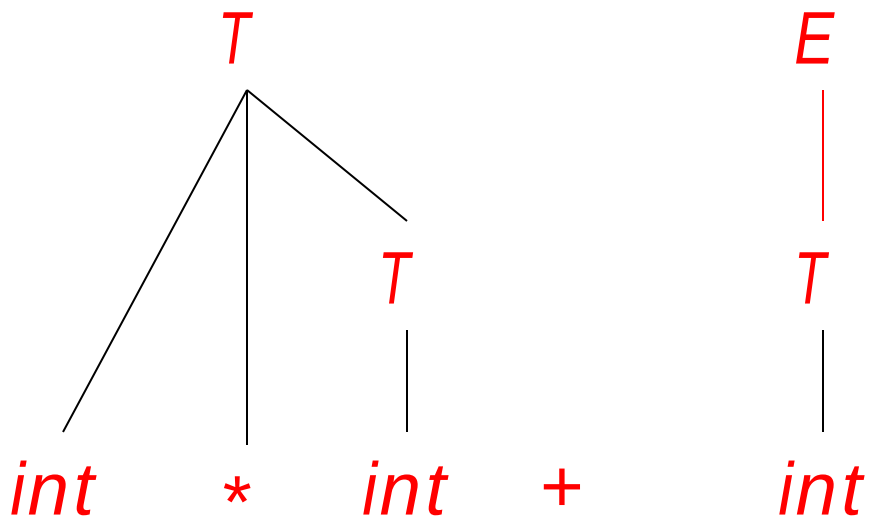
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

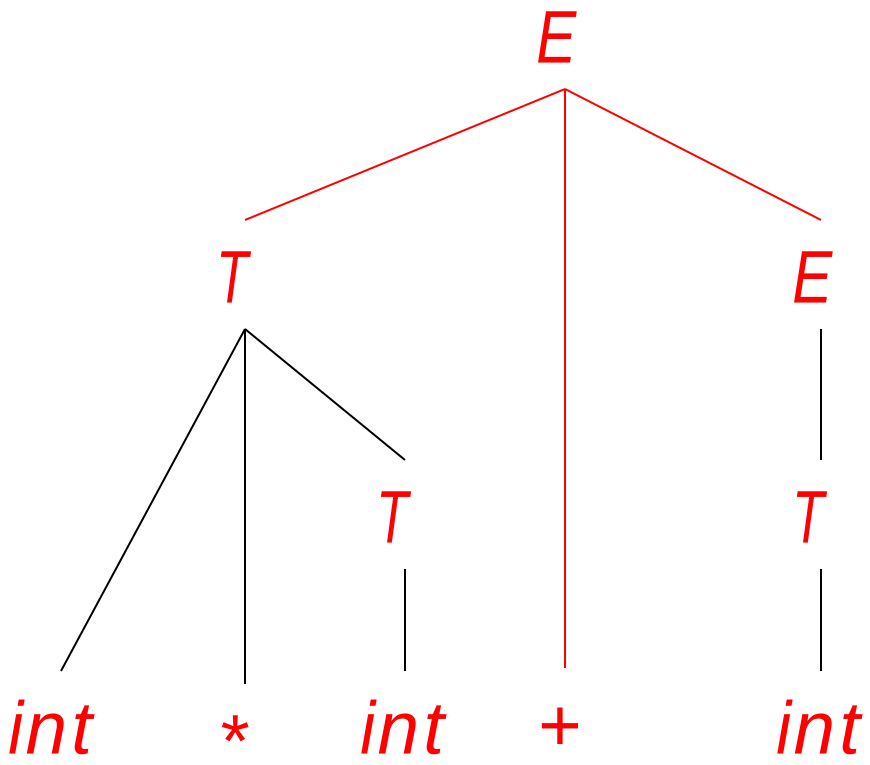
int * T + int

T + int

T + T

T + E

E

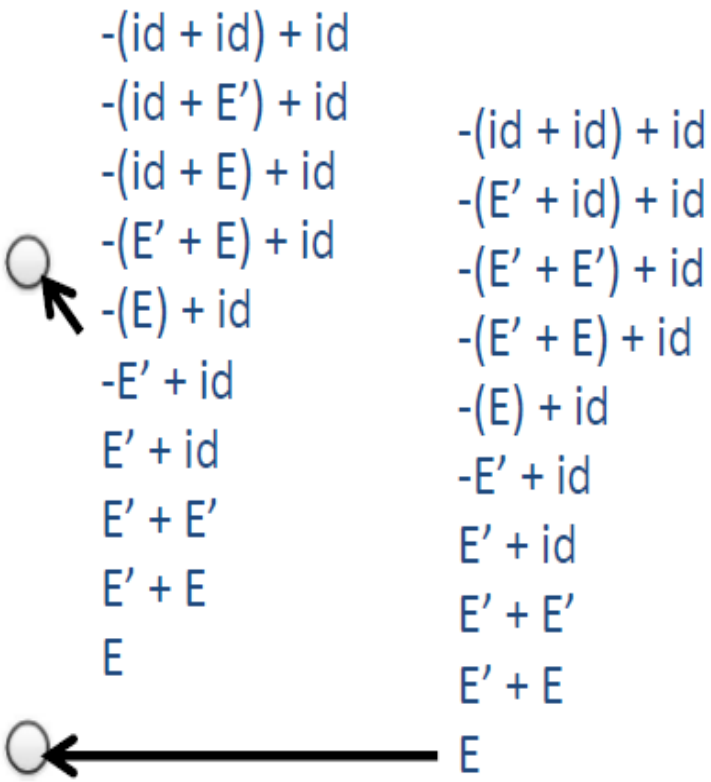


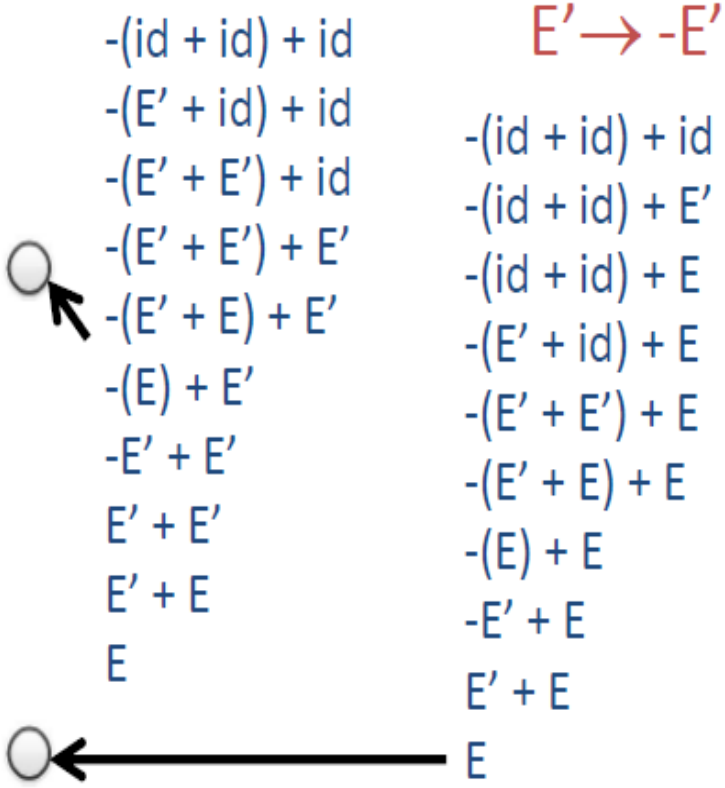
Quiz

For the given grammar, what is the correct series of reductions for the string: -(id + id) + id

$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$





Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

By using reductions instead of productions

Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why?

Because $\alpha X \omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

A Trivial Bottom-Up Parsing Algorithm

Let I = input string

repeat

pick a non-empty substring β of I

where $X \rightarrow \beta$ is a production

if no such β, backtrack

replace one β by X in I

until I = "S" (the start symbol) or all possibilities are exhausted

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a |
  - The | is not part of the string
- Initially, all input is unexamined $|x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- *Shift:* Move | one place to the right
  - Shifts a terminal to the left string

ABC|xyz $\Rightarrow$ ABCx|yz

Reduce

- Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$

The Example with Reductions Only

int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

A Shift-Reduce Parse in Detail (1)

|int * int + int

int * *int* + *int*

↑

A Shift-Reduce Parse in Detail (2)

|int * int + int

int | * int + int

A Shift-Reduce Parse in Detail (3)

|int * int + int

int | * int + int

int * | int + int

A Shift-Reduce Parse in Detail (4)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * *int* + *int*

A Shift-Reduce Parse in Detail (5)

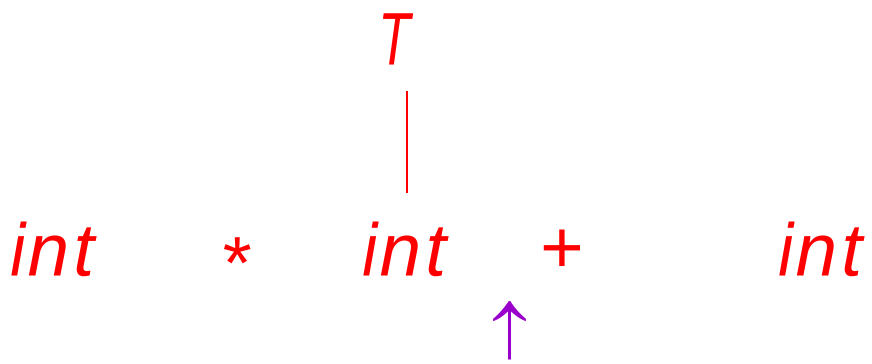
|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

|int * int + int

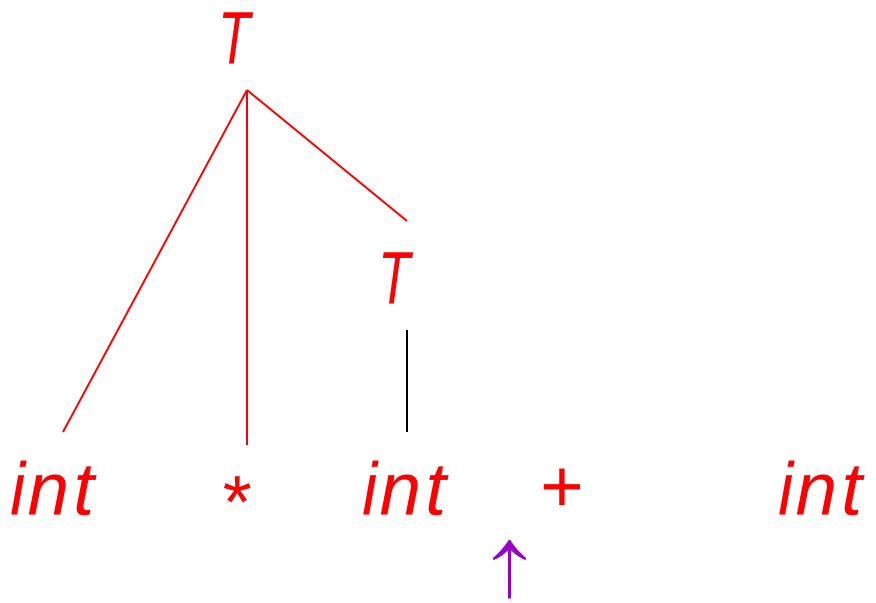
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

|int * int + int

int | * int + int

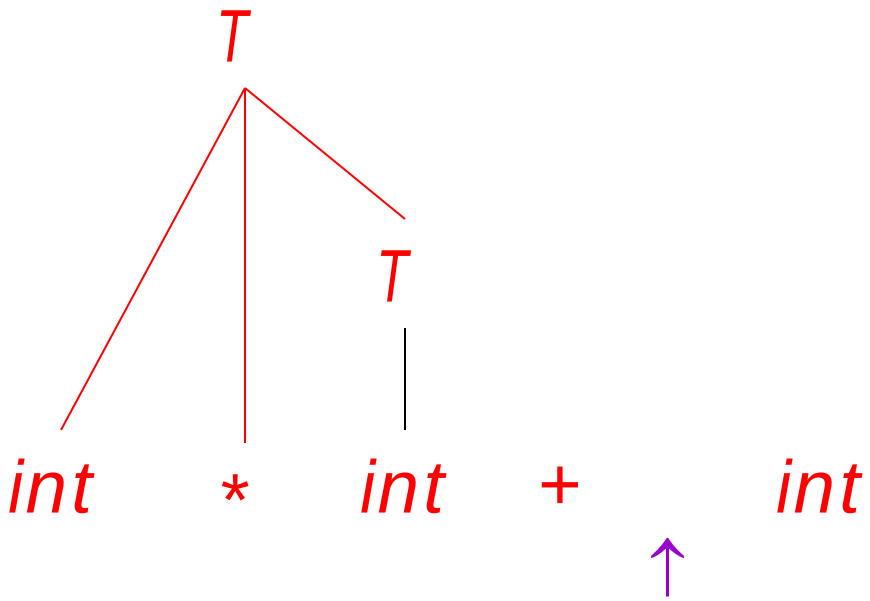
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



A Shift-Reduce Parse in Detail (8)

|int * int + int

int | * int + int

int * | int + int

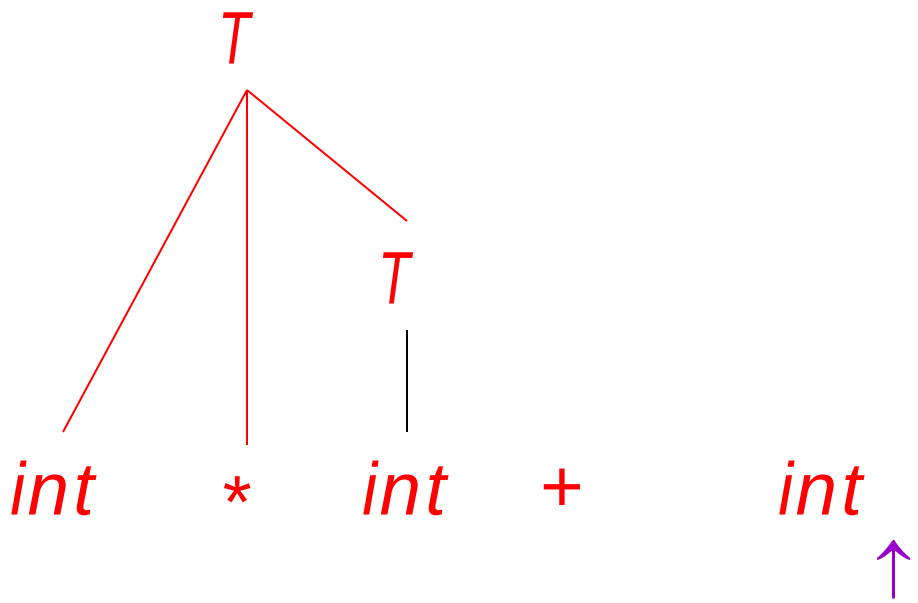
int * int | + int

int * T | + int

T | + int

T + | int

T + int |



A Shift-Reduce Parse in Detail (9)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

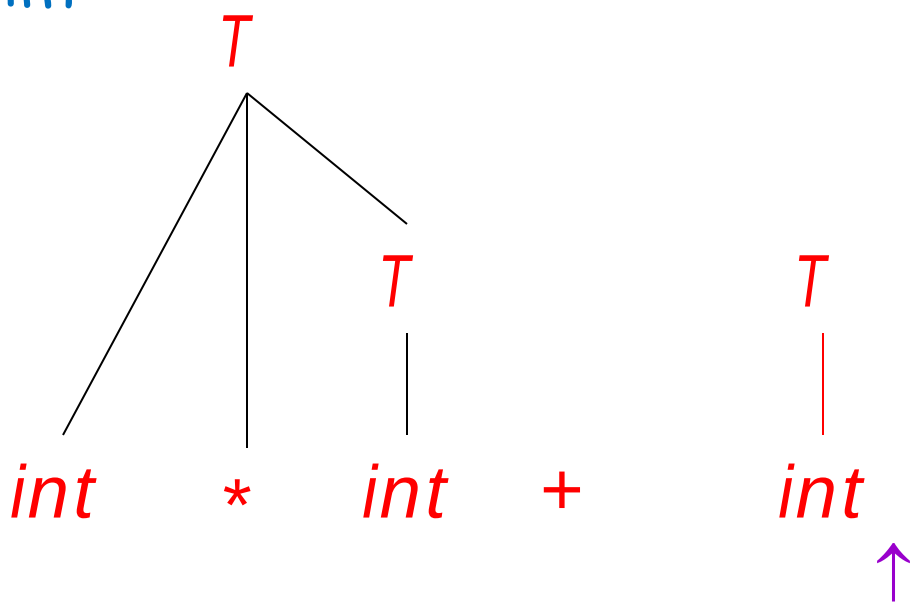
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

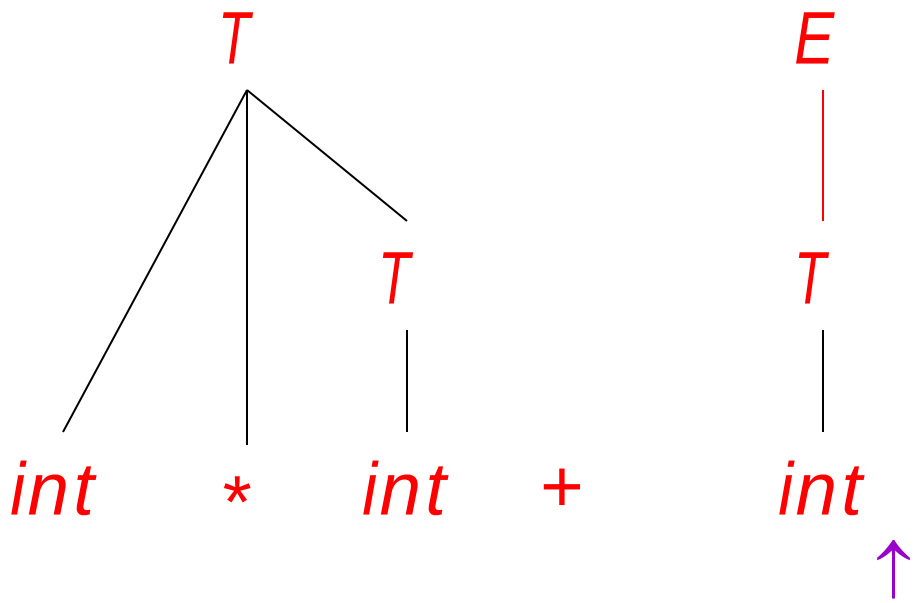
T | + int

T + | int

T + int |

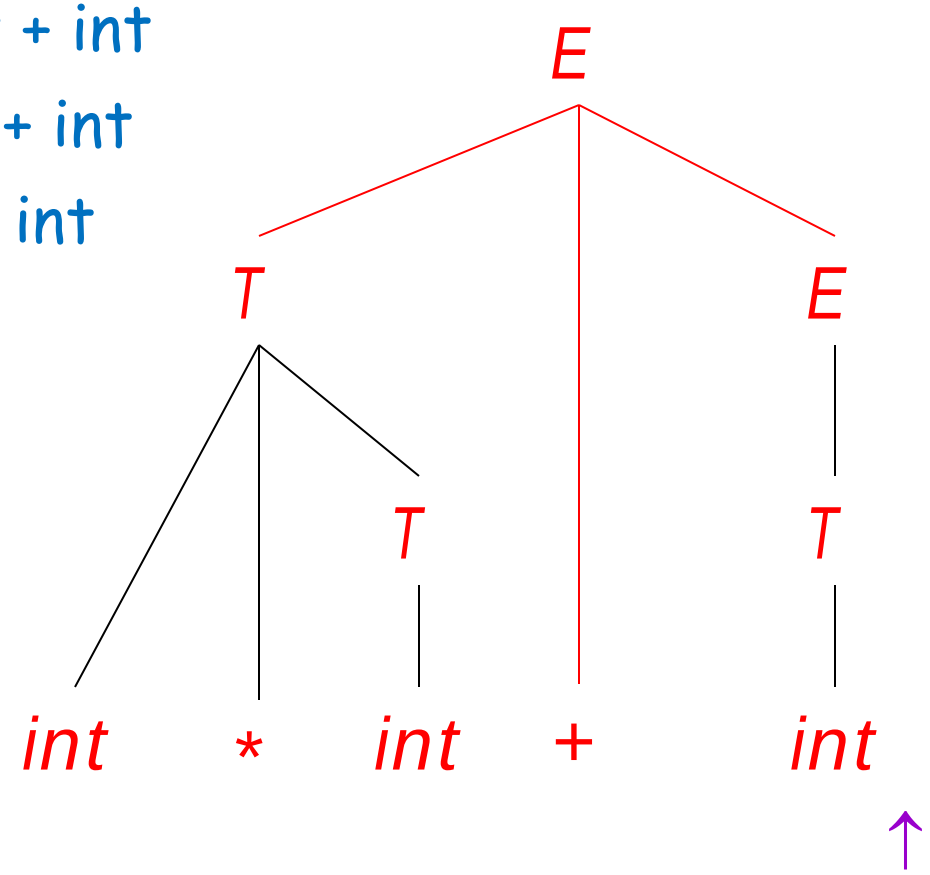
T + T |

T + E |



A Shift-Reduce Parse in Detail (11)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |



Note about the implementation

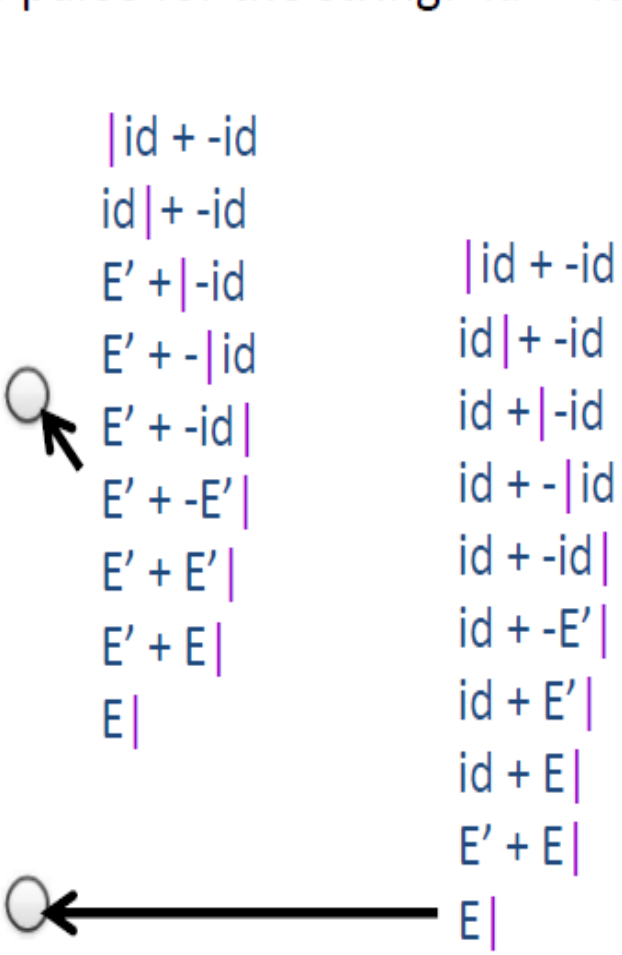
- The example just shows that there exists a sequence of shift and reduce moves that succeed in parsing the input
- However, it does not explain how we know whether to shift or reduce

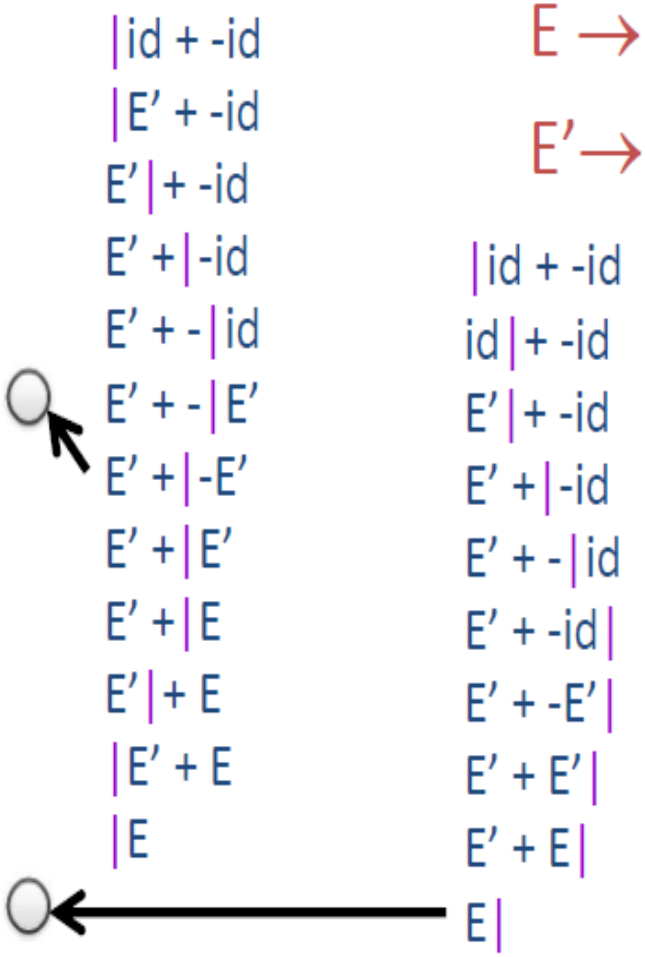
Quiz

For the given grammar, what is the correct shift-reduce parse for the string: id + -id

$E \rightarrow E' \mid E' + E$

$E' \rightarrow -E' \mid id \mid (E)$





Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- If it is legal to shift or reduce, there is a *shift-reduce* conflict
- If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict

Summary: Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Read one input token and move the vertical bar one place to the right

ABC|xyz $\Rightarrow$ ABCx|yz

Reduce

Replace the right hand side of the production …the sequence to the left of the vertical bar…..by the left hand side of the production

- Cbxy|ijk $\Rightarrow$ CbA|ijk

The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce
  - pops o or more symbols off of the stack
    - production rhs
  - pushes a non-terminal on the stack
    - production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:

  $E \rightarrow T + E \mid T$

  $T \rightarrow int * T \mid int \mid (E)$

- Consider step int | * int + int
  - We could reduce by $T \rightarrow int$ giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E
    - There is no production starts with T*

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol
- We do not want to reduce just because we have the right-hand side of a production on top of the stack
- Assume a rightmost derivation

$$S \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega$$

- Then $X \rightarrow \beta$ in the position after α is a *handle* of $\alpha\beta\omega$

Handles (Cont.)

- Handles formalize the intuition
  - A handle is a string that **can be reduced and also allows further reductions back to the start symbol** (using a particular production at a specific spot)
- We only want to reduce at handles
- Note: We have said what a handle is, not how to find handles

Important Fact #2

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside

Why?

- True initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the | never move left
- Bottom-up parsing algorithms are based on recognizing handles

Quiz

Given the grammar at right, identify the handle for the following shift-reduce parse state: E' + -id | + -(id + id)

$E \rightarrow E' \mid E' + E$

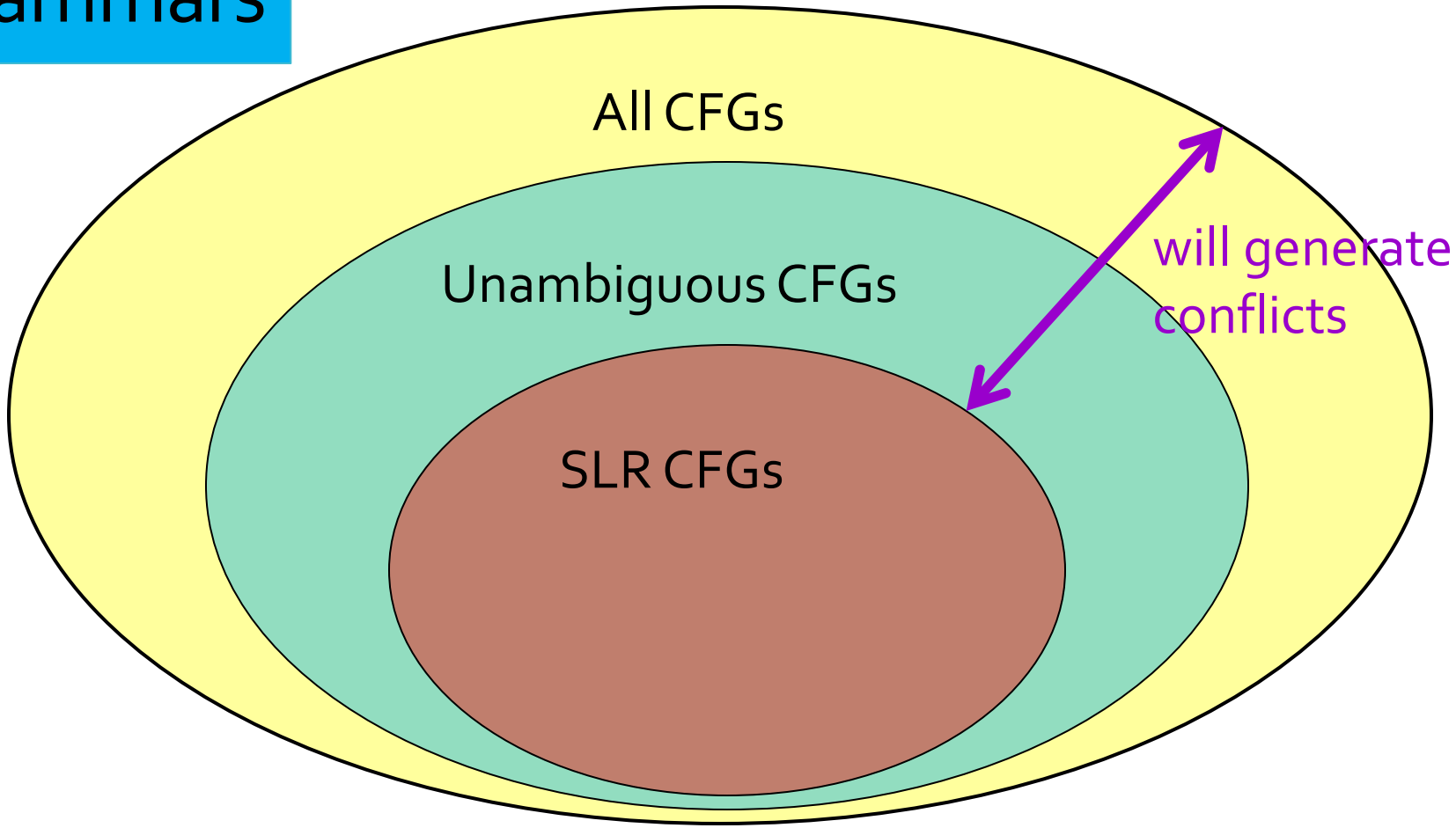
$E' \rightarrow -E' \mid id \mid (E)$

- E' + -id
- id
- -id
- E' + -E'

Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars

Grammars


All CFGs
Unambiguous CFGs
SLR CFGs
will generate conflicts

Viable Prefixes

- It is not obvious how to detect handles
- At each step the parser sees only the stack, not the entire input; start with that . . .

α is a viable prefix if there is an ω such that $\alpha|\omega$ is a state of a shift-reduce parser

What does this mean?

- A viable prefix does not extend past the right end of the handle
- It's a viable prefix because it is a prefix of the handle
- As long as a parser has viable prefixes on the stack no parsing error has been detected

Important Fact #3

Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language

Important Fact #3 (Cont.)

- Important Fact #3 is non-obvious
- We want to show how to compute automata that accept viable prefixes

Items

- An item is a production with a "." somewhere on the rhs
- The items for $T \rightarrow (E)$ are
  - $T \rightarrow .(E)$
  - $T \rightarrow (.E)$
  - $T \rightarrow (E.)$
  - $T \rightarrow (E).$

Items (Cont.)

- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow .$
- Items are often called "LR(0) items"

Intuition

- The problem in recognizing viable prefixes is that the stack has only pieces of the rhs of productions
  - If it had a complete rhs, we could reduce
- These pieces are always *prefixes* of rhs of productions

Example

Consider the input (int)

- Then (E|) is a state of a shift-reduce parse
- (E is a prefix of the rhs of T → (E)
  - Will be reduced after the next shift
- Item T → (E.) says that so far we have seen (E of this production and hope to see) before we can perform a reduction

An Example

Consider the string (int * int):

(int *|int) is a state of a shift-reduce parse

"(" is a prefix of the rhs of $T \rightarrow (E)$

"ε" is a prefix of the rhs of $E \rightarrow T$

"int *" is a prefix of the rhs of $T \rightarrow int * T$

An Example (Cont.)

The "stack of items"

$T \rightarrow (.E)$

$E \rightarrow .T$

$T \rightarrow int * .T$

Says

- We've seen "(" of $T \rightarrow (E)$
- We've seen ε of $E \rightarrow T$
- We've seen int * of $T \rightarrow int * T$

Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs's of productions, where
- Each sequence can eventually reduce to part of the missing suffix of its predecessor

Thanks